FROM THE BENCH

Long-Range, Low-Power Wireless Communications (Part 1)

Trading Throughput for Distance

If you find that your next design project's requirements exceed the specifications of the wireless technology you are presently using, you might want to consider LoRa. In this series, Jeff introduces LoRa technology and discusses some of its advantages over older competitors.

By Jeff Bachiochi

COLUMNS

As a young man, Marconi created a storm monitor—a rudimentary receiver—used to detect lightning. He extended this by recreating his own storm on demand, sending broad spectrum electromagnetic (EM) waves propagating from a spark gap, a simple transmitter. While these transmissions were extremely wide band, they bootstrapped the development of radio telegraphy (Morse code) and telephony (voice) within the next decade. Now, 100 years later, you can talk to virtually anyone in the world with a portable wireless device that fits in your pocket.

I still listen to commercial radio while driving in my Jeep, which has an AM/FM radio. The familiar digits of a station's frequency reassure me that I'll be hearing a good mix of new and old tunes. This number is a radio station's carrier frequency. It's just a small slice of the FCC's allocated spectrum for the FM broadcast band from 88.0 to 108.0 MHz. Station carriers are allocated every 200 kHz inside this band. Initially, audio was used to directly modulate the carrier for a monaural transmission requiring much less that the allocated bandwidth of 200 kHz. Today's radio stations have found ways to use this added bandwidth, but that's a different story. This one relates to the fact that these radio frequencies can pass right through our homes.

Practical radio transmission systems are mostly line-of-sight propagation or radio waves that travel in a straight line from the transmitting antenna to the receiving antenna. This process applies to garage door openers, cell phones, cordless phones, walkie-talkies, wireless networks, commercial radio and television broadcasting, and radar. Line-of-sight transmission on the Earth's surface is limited to the distance to the visual horizon, a maximum of about 40 miles. It does not necessarily require a cleared sight path; at lower frequencies, radio waves can pass through building walls and foliage.

Many of the wireless communication mediums used today are part of IEEE 802. This is a family of standards from the Institute of Electrical and Electronics Engineers that deals with local area and metropolitan area networks. These include working groups that focus on wired and wireless connection types. You may be most familiar with some of the 802 wireless groups like 802.11 (Wi-Fi), 802.15.1 (Bluetooth), 802.15.4 (ZigBee, MiWi), 802.15.6 (Body Area Network), and 802.15.7 (Visible Light Communications). As carrier frequencies rise above 1 GHz, you'll find these technologies begin to have penetration issues, as you can see in **Figure 1**.[1] It's not a big deal when you are dealing with short transmission distances.

The FCC has a tough job. The usable spectrum is limited. It is their job to allocate the available space. Everyone wants a piece of the action. To ensure everyone plays well together, the FCC sometimes must impose additional restrictions on top of who can use which parts of the spectrum. These might include maximum power radiated or transmission time limits. Many bands are licensed. Some are open, but you must still play by the rules. One of the open bands is the ISM (Industrial, Scientific, and Medical) band, 915 MHz (902 to 928 MHz). In the United States, this ISM band is also shared with error-tolerant communications applications such as wireless sensor networks. My wireless house phone operates in this region.

We are beginning to see some technologies pick up on this. One new player is LoRa (Long Range). In this article, I'll introduce the technology and then present an interesting project—a peer-to-peer network using the LoRa technology. I'll cover the circuitry and explain how the hardware works.

LoRa SPECTRUM

Unlike many other technologies, the slots that make up the LoRa spectrum are divided into a number of different bandwidths. The 915-MHz (US) LoRa band, shown in **Figure 2**, has 64, 125 kHz, and 8, 500 kHz uplink channels and 8, 500 kHz down link channels. Note uplink channels share the same space. A transmission may select either bandwidth allocation to use and trade off SNR for throughput. A transmission always begins with channel 0. This allows a receiver to catch the transmission before hopping channels in a predetermined sequence. LoRa uses chirp spread spectrum (CSS) and is a technique that uses wideband linear frequency modulated "chirp" pulses to encode information. A chirp is a rising (or falling) sinusoidal signal frequency that varies over time. The chirp's frequency varies proportional to the bandwidth of the allocated slot. One or more chirps can reside within a single time slot (frequency hop).

The data throughput is fixed by a number of factors in the LoRa specification, the bandwidth (BW), the spreading factor (SF), and the coding rate (CR). In our case (915-MHz band), the uplink BW is fixed at either 125 or 500 kHz. The SF is the number of chips or divisions that make up a symbol. A symbol is sent for each 1 bit of data and is a power of 2—in this case, between 6 and 12. The CR is an amount of forward error correction information added to each symbol, here between 1 and 4 or 0.25 to 1 extra bit.

The actual data bit count we began with has suddenly been expanded by the SF and CR into a much larger number of transmission data chips. The transmission data modulates the chirping carrier at a rate of 1 chip per second per 1 Hz (here, that's 125 kbps). At a maximum SF and CR, that's about 293 actual data bits per second (approximately 36 bytes). All this redundancy, allows for the best-case SNR, which should mean a receiver can more easily find and decode the transmission, resulting in an increase in range.

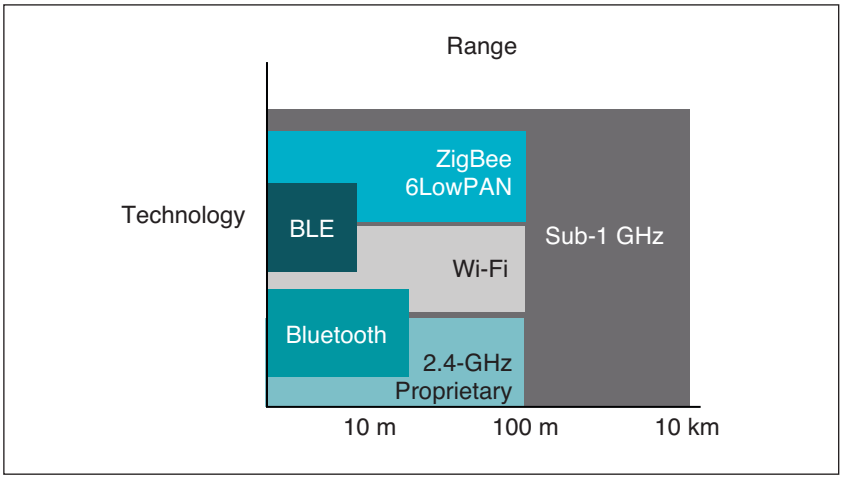

FIGURE 1
Frequency penetration

DATA PACKET

Before we look at the radio's register set and initialize it, we should look at the elements of the packet sent by the LoRa radio. (The register set for the Semtech radio chip is available on the *Circuit Cellar* FTP site. The register set is multipurpose—that is, it's register usage depends on the type of modulation—OOK/FSK/LoRa—identified in the Operational Mode Register.) There are a few decisions that should be made in order to set the configuration correctly. The LoRa packet is made up of a preamble, optional header, and payload. The preamble contains special symbols that allow the receiver to recognize it as a packet start. The length of preamble is programmable but must contain at least the syncword and sync downchirp. In our case, we'll use the default of eight, plus program the LoRa syncword 0x12 into register 0x39 (RegSyncWord). Note: It looks like all modulation is up chirps except the final sync in the preamble.

The ImplictHeaderMode bit found within the RegSymbTimeoutMsb register can be set to Explicit Mode. This adds an optional header when the transmitter

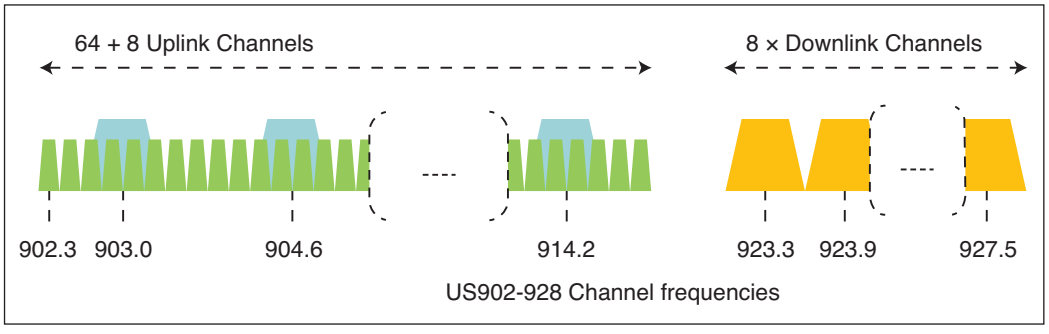

FIGURE 2
LoRa 915-MHz band

Operating Mode	DIOx Mapping	DIO5	DIO4	DIO3	DIO2	DIO1	DIO0
ALL	00	ModeReady	CadDetected	CadDone	FhssChangeChannel	RxTimeout	RxDone
	01	ClkOut	PllLock	ValidHeader	FhssChangeChannel	FhssChangeChannel	TxDone
	10	ClkOut	PllLock	PayloadCrcError	FhssChangeChannel	CadDetected	CadDone
	11	-	-	-	-	-	-

TABLE 1
The radio's general-purpose I/Os can be configured to "map" particular status to the physical outputs. The 2-bit DIOx mapping value is written to the appropriate mapping registers (0x40 for DIO0:3 and 0x41 for DIO4:5).

wants to tell the receiver; how many bytes to expect, what Coding Rate (CR) is being used for Forward Error Correction (FEC), and an optional 16-bit Cyclic Redundancy Check (CRC) for the payload. This can be eliminated if both transmitter and receiver have matching values manually configured on each side. This effectively shortens the packet size but offers less flexibility of the packet. For this project, we'll be using the Explicit Mode. This requires configuration of registers 0x1D (RegModemConfig1), 0x1E (RegModemConfig2), and 0x26 (RegModemConfig3). Also, the SF, CR, and other information must be written so the transmitter can assemble the header information correctly.

The payload length is a maximum of 255 bytes; however, there is a maximum packet time of 400 ms, as defined by the FCC for the US 902–928 band (915 MHz). This will limit the available payload length to a lower value, especially when using higher SF. Payload data will be written into a FIFO through register 0x00 (RegFifo) to send it. On the receiver, the payload data will be read through this register. The payload is handled by an internal 64-byte FIFO. Data pointers are automatically adjusted as the data is written in by the user and transmitted out by the radio and received by the radio and read out by the user.

You'll note (see **Table 1**) that the radio offers six I/Os that can be configured to indicate the status of various functions like PayloadReady (in RX mode) or PacketSent (in TX mode). These can be mapped to pins DIO0:5 using two registers, RegDioMapping1 0x40 (D0:3) and RegDioMapping2 0x41 (D5:4).

LoRa VS LoRaWAN

Semtech Corp. is responsible for the proprietary LoRa radio specification. They are also a founding member of the LoRa Alliance (www.lora-alliance.org), whose mission is to standardize Low Power Wide Area Networks (LPWAN) and enable the LoRa protocol (LoRaWAN). Semitech's product line includes wireless RF transceivers. A slice of that pie is devoted to LoRa radios. The LoRa radio is the mechanism for sending data using the LoRa technology. This should not to be confused with LoRaWAN, which is a standard for building a LPWAN upon the LoRa technology.

The LoRa radio specification is the mechanism used to transmit a payload over the air. Other than the initial structure or wrapper used around the payload, it does not define the payload in any way. LoRaWAN is a Media Access

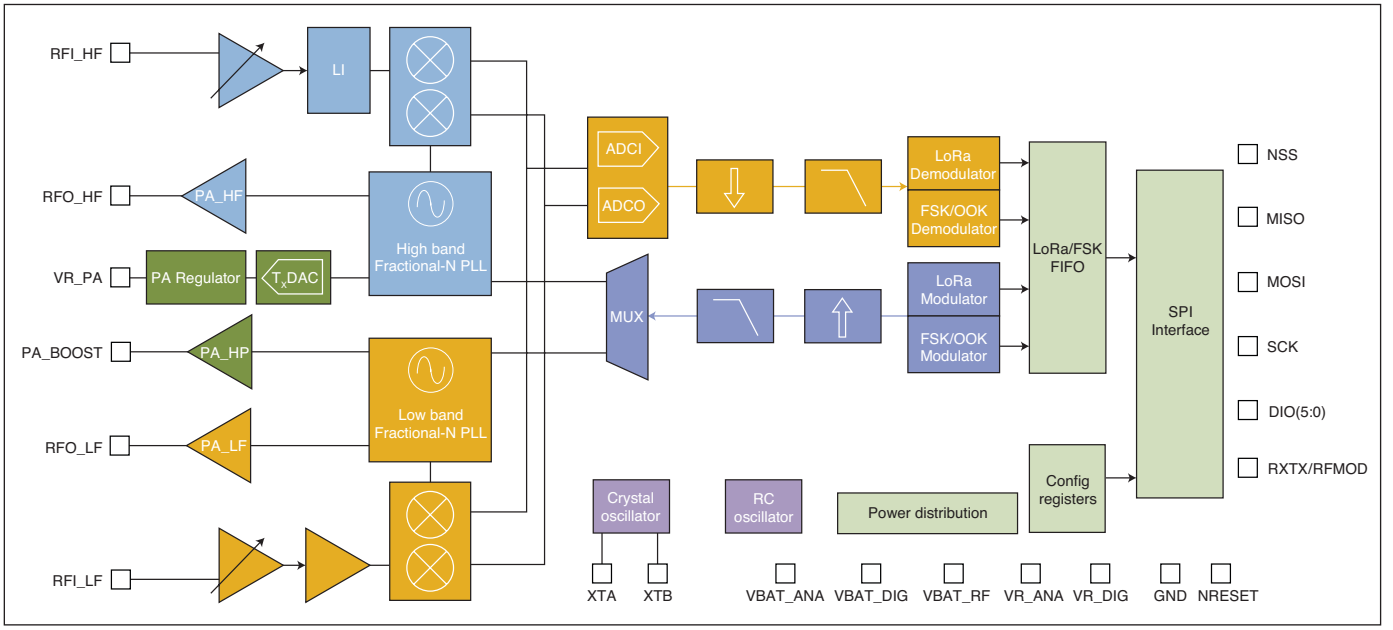

FIGURE 3
A diagram of the radio chip

Control (MAC) protocol for wide area networks. It allows low-powered devices to communicate with Internet-connected applications over LoRa wireless connections. It is one of the protocols implemented on top of LoRa technology. It allows a cloud-based application to collect and direct communications between itself and distant end points.

LoRaWAN uses the LoRa wireless technology in the end point to gateway portion of the WAN. Gateways connect LoRa wireless nets to cloud control via internet connections. LoRaWAN is beyond the scope of this project. This project will investigate the use of LoRa technology for peer-to-peer wireless networking where an increase in transmission distance is required. I will focus my attention on the LoRa technology, which can be used on its own for smaller scale peer-to-peer networking. The hardware I chose to begin with is the Adafruit Feather 32u4 Radio with RFM95W Module, which is a $25 small form factor Arduino with LoRa-compatible radio. The RFM95W is a radio module made by HopeRF Electronic. The Feather is available with either a 433-MHz (Europe) or 868/915-MHz (US) radio. You can find support libraries for the LoRa radio, but I'll be using the SPI directly to read and write radio registers.

The RFM95W radio module uses Semtech's SX127x family LoRa transceiver. The SPI requires a few I/O: CS output (SPI Chip Select), Reset output, and Interrupt input for the radio status. Let's start with a look inside the transceiver. **Figure 3** shows the function blocks of the radio module. You'll notice that this radio is very flexible, because it can perform the normal FSK and OOK modulations as well as LoRa within its selected frequency band. Refer to **Table 2** for the register set for the radio. You'll note some register functions change depending on the mode set in the radio's Operation Mode Register (0x01 in the table), which defaults to FSK at power-up.

While the Feather32U4 has pad provisions for a uFL (antenna) connector, I chose to use a

RegOpMode (0x01)	Bits	Variable Name	Mode	Default value	FSK/OOK Description
	7	LongRangeMode	r	B'0'	0 - FSK/OOK Mode
					1- LoRaTM Mode
					This bit can be modified only in Sleep mode.
					A write operation on other device modes is ignored
	5-Jun	ModulationType	rw	B'00'	Modulation scheme:
					00 - FSK
					01 - OOK
					10:11 - reserved
	4	reserved	r	B'0'	reserved
	3	LowFrequencyModeOn	rw	B'1'	Access Low Frequency Mode registers (from address 0x61 on)
					0 - High Frequency Mode (access to HF test registers)
					1 - Low Frequency Mode (access to LF test registers)
	2:00	Mode	rw	B'001'	Transceiver modes
					000 - Sleep mode
					001 - Stdby mode
					010 - FS mode TX (FSTx)
					011 - Transmitter mode (Tx)
					100 - FS mode RX (FSRx)
					101 - Receiver mode (Rx)
					110 - reserved
					111 - reserved

TABLE 2
The Operational Mode register sets the basic function of the radio to either FSK/OOK or LoRa mode.

Name (Address)	Bits	Variable Name	Mode	Reset	LoRaTM Description
Reg Irq Flags (0x12)	7	RxTimeout	Read/clear	0x00	Timeout interrupt: a write operation clears IRQ
	6	RxDone	Read/clear	0x00	Packet reception complete interrupt: a write operation clears IRQ
	5	PayloadCrcError	Read/clear	0x00	Payload CRC error interrupt: a write operation clears IRQ
	4	ValidHeader	Read/clear	0x00	Valid header received in Rx: a write operation clears IRQ
	3	TxDone	Read/clear	0x00	FIFO Payload transmission complete interrupt: a write operation clears IRQ
	2	CadDone	Read/clear	0x00	CAD complete: write to clear: a write operation clears IRQ
	1	FhssChangeChannel	Read/clear	0x00	FHSS change channel interrupt: a write operation clears IRQ
	0	CadDetected	Read/clear	0x00	Valid Lora signal detected during CAD operation: a write operation clears IRQ

TABLE 3
The individual interrupt status is available through the IRQ resister. We'll be mainly interested in the RXDone and TXDone bits.

3" piece of wire as a quarter wave whip antenna for my initial experiments. No additional connections other than the USB, which is used for powering the board and serial debugging, are required to get started.

ARDUINO SKETCH

Most Arduino sketches contain at least two functions, `setup()` and `loop()`. Prior to the `setup()` function, you must declare/define those external libraries, constants, and variables of which you will call upon. I'll be using the standard SPI and Wire libraries for communicating with peripherals, but not any libraries that mask how those peripherals might be used. All the registers are defined with names that suggest their use, to make the code more readable.

Since the radio module has outputs that can be used as interrupts, we need to make sure that those variables are declared with the "volatile" qualifier. One variable I use in every sketch is `int debug`. This gives me 16 bits I can use to enable/disable `print` statements that I use to show the present values of variables. By qualifying a print statement with `if((debug & <bit value> == <bit value>)`, a print statement can be enabled or disabled from the serial output stream by setting or clearing the appropriate bits in the debug variable. So far, there has not been a good debugger for the Arduino IDE. If the code crashes, there are no clues to what's going on. It behooves the user to add some kind breadcrumb trail to help follow the execution.

The `setup()` function is the place to start any libraries or initialize any peripherals. I'm using the USB for serial debugging and SPI to communicate with the radio module. The feather32u4 has a few permanent connections to the radio module, and since all I/O defaults to inputs, we must configure the SPI's chip select (CS) as an output in an initialized HIGH state. Since this project require at least two radios for communications, I'll be using the same sketch to operate either in a client or server mode and simplify the project. This choice could be set with a physical jumper. In this case, I'll just set a Boolean variable at compile time to determine the function. This is part of the sign-on message and will display which function has been assigned to a board.

Many Arduino sketches never make use of an interrupt. The radio module can map six DIO pins to an internal status state, which might be used as interrupt sources. However, most Arduino boards have a limited number of inputs that can be defined as external interrupts. We'll use one of these outputs to signal when a selected function has completed. To use an external interrupt in an Arduino sketch, we must define its connection in `setup()` using the `attachinterrupt()`

circuitcellar.com/ccmaterials

REFERENCE

[1] G. Reiter, "Wireless connectivity for the Internet of Things," Texas Instruments, 2014, www.ti.com/lit/wp/swry010/swry010.pdf.

SOURCES

Feather 32u4 Bluefruit LE
Adafruit | www.adafruit.com

RFM95W 868/915Mhz RF Transceiver Module
HOPE Microelectronics | www.hoperf.com

function. This assigns an input pin to interrupt routine and also defines the kind of operation that will cause the interrupt (a logic level or change in state).

INTERRUPTS

In the `setup()`, we just described what it take to indicate that an interrupt event has taken place. Now, we need to define what happens when the event occurs with a function (named in `setup()`'s `attachinterrupt()` function). We are using one interrupt input to signal multiple potential radio events. We can determine what event caused the interrupt by reading the radio's "IRQ Flags" register (see **Table 3**).

If we've enabled the Receive mode and either an RXTimeout or PayloadCRCError has occurred, then nothing (good) has been received and we can set Idle mode and leave. If the RXDone bit is set, then we want to read the received payload from the FIFO into our buffer, flag it as "good," set Idle mode, and leave. If we've transferred a payload to send from the buffer to the FIFO and enabled the Transmit mode, then the radio will wrap our payload up and transmit it. When it has been sent, the TXDone bit is set. We don't need to do anything since the work is done, except to set Idle mode and leave.

It's worth noting here that the radio provides another function that may be of interest. The CAD mode can be used to check the radio for channel activity. In a busy environment, it may be advantageous to see if anyone else is transmitting, so you don't trample their communication.

LOOP()

We finally arrive at the point where we can put all of this groundwork to good use. The main `loop()` will simply divide the application into one of two functions—that of client or server (based on my Boolean variable `IamAServer`). A client will transmit a packet to the server and look for some response. A server will monitor for a client's request and respond to it. **Photo 1** shows the set up for this column executing the Arduino sketch. You might picture an application where some sensor array is beyond the immediate area. Most wireless networks are somewhat limited in maximum distance—say, within the extents of your home. While you might be able to see a neighbor's Wi-Fi network, its signal strength will most likely be weak and unreliable.

To test the LoRa radios, we'll send a simple message between endpoints and look for confirming responses. One of the reasons for selecting the Feather32u4 is the inclusion of a Lithium cell charger that allows a module to run on one Lithium cell and be recharged via the USB connection. This enables one unit to become mobile so I can do some range testing. To aid in implementation, I will add a few items to this circuit, a couple of push buttons to adjust the radio's TX power, and a small display to show the test results. I'll be adding this to one radio so it can become a mobile test bed.

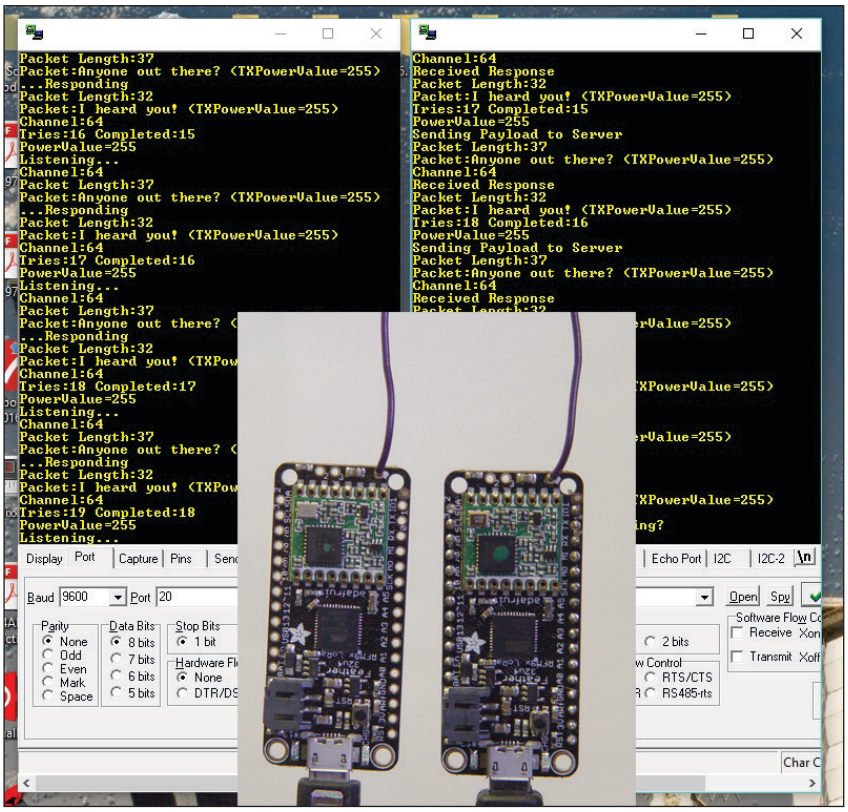

PHOTO 1

These are the Feather32U4 boards I'm using for this project. Twin copies of Realterm are executing on my PC. These are connected through USB cables to the client and server PCBs. You can see the ongoing conversations taking place.

Next time, before we proceed with some range testing, we'll look at the simple mechanism for implementing data transactions and what's required to support a small I²C 128 × 64 pixel OLED display. This will provide up to eight rows of data in less than 1 square inch, at a parts cost of less than $10, and an increase in the current budget of only around 10 mA!

ABOUT THE AUTHOR

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for *Circuit Cellar* since 1988. His background includes product design and manufacturing. You can reach him at jeff.bachiochi@imaginethatnow.com or at www.imaginethatnow.com.

FROM THE BENCH

Long-Range, Low-Power Wireless Communications (Part 2)

Put LoRa to Work

COLUMNS

This series serves as an introduction to LoRa technology and some of its advantages over older competitors. This month, before covering range testing, Jeff considers the simple mechanism for implementing data transactions and what's required to support a small I²C 128 × 64 pixel OLED display.

By Jeff Bachiochi

The tendency regarding communications has been that more and faster is better. Most wireless widgets are dealing with relatively short distances: mouse/keyboard to a PC, clicker (IR remote) to a TV, or RFID tags to point of purchase registers. We enjoy the freedom wireless connections give us. Since these technologies are based on low power and limited distance, they often can't be used where there is any extended distance or physical barriers blocking the signal.

A new standard for long-range or low-power radio (LoRa) has been in the news lately. Using a sub-gigahertz frequency, LoRa not only provides a deeper penetration in dense urban/indoor regions, but also reaches farther into rural areas. The technique involves trading increased distance for higher speeds. This is meant to fill a present gap where remote sensors must report their status to others at some distance away.

You might've also heard of LoRaWAN, which is a protocol specification built on top of the LoRa technology and developed by the LoRa Alliance. LoRa technology uses the unlicensed radio spectrum in the Industrial, Scientific, and Medical (ISM) bands. LoRaWAN enables wide area communication between remote sensors and gateways connected to the network using LoRa hardware and software for bidirectionally secure communication. I won't cover the LoRaWAN protocol in this article. I'll merely explore LoRa technology in a peer-to-peer connection.

Semtech Corp.—a leading supplier of high-quality analog and mixed-signal semiconductor products—is the innovator of the LoRa technology with its SX1272 RF transceiver. HOPERF Micro-electronics Co. manufactures wireless radio core components and IoT sensors, and it combines Semtech technology with the external components required to produce a LoRa radio module, the RFM95W. The module's solder points gives access the transceiver via an SPI. Adafruit took the RFM95W and added it to an Atmel ATmega32u4 board, the Feather 32u4. As a result, we have complete domination over the LoRa module using the popular Arduino IDE.

For the project described in this article, I used two Feather 32u4s board to form a peer-to-peer network using the LoRa technology. Last month, I introduced the circuitry and discussed the innards of the Semtech device and how it and the ATmega work together. Please review that article for a closer look at LoRa technology. Now we'll put that to work. Because much of the support for the RFM95 is applicable for both a client and a server, it makes sense to use a single application (Arduino sketch) for both situations. Let's start with a look at the sketch elements.

BIRD'S EYE VIEW

The RFM95W communicates with the ATmega via four SPI lines. Additional radio outputs offer status to the processor. When any of these are used as an interrupt trigger, you must provide an Arduino ISR routine for each interrupt input. I've enabled one interrupt and provided `RFM_ISR` as the routine that will be called whenever the RFM95W signals an event. An interrupt routine is independent of the main program flow, that is, it will take precedent over normal code execution. As you can see in **Figure 1**, when this occurs, the `irq-flags` are read from the RFM95W. These flags, along with the present mode set by the main program (RX, TX, IDLE, etc.), will determine what task is performed by the interrupt routine.

If the mode is RX, then we are waiting for the RFM95W to receive a packet. When the packet arrives, it is read from the RFM95W and put into a buffer `Packet[]` and the `rxBufValid` flag is set. If the mode is TX, then we've already written the buffer `Packet[]` to the RFM95W and we're just waiting for confirmation that the packet has been sent. The CAD mode doesn't send or receive any data it just lets you know if there is any RF activity. You may want to make sure that the "airspace is free" before sending any data.

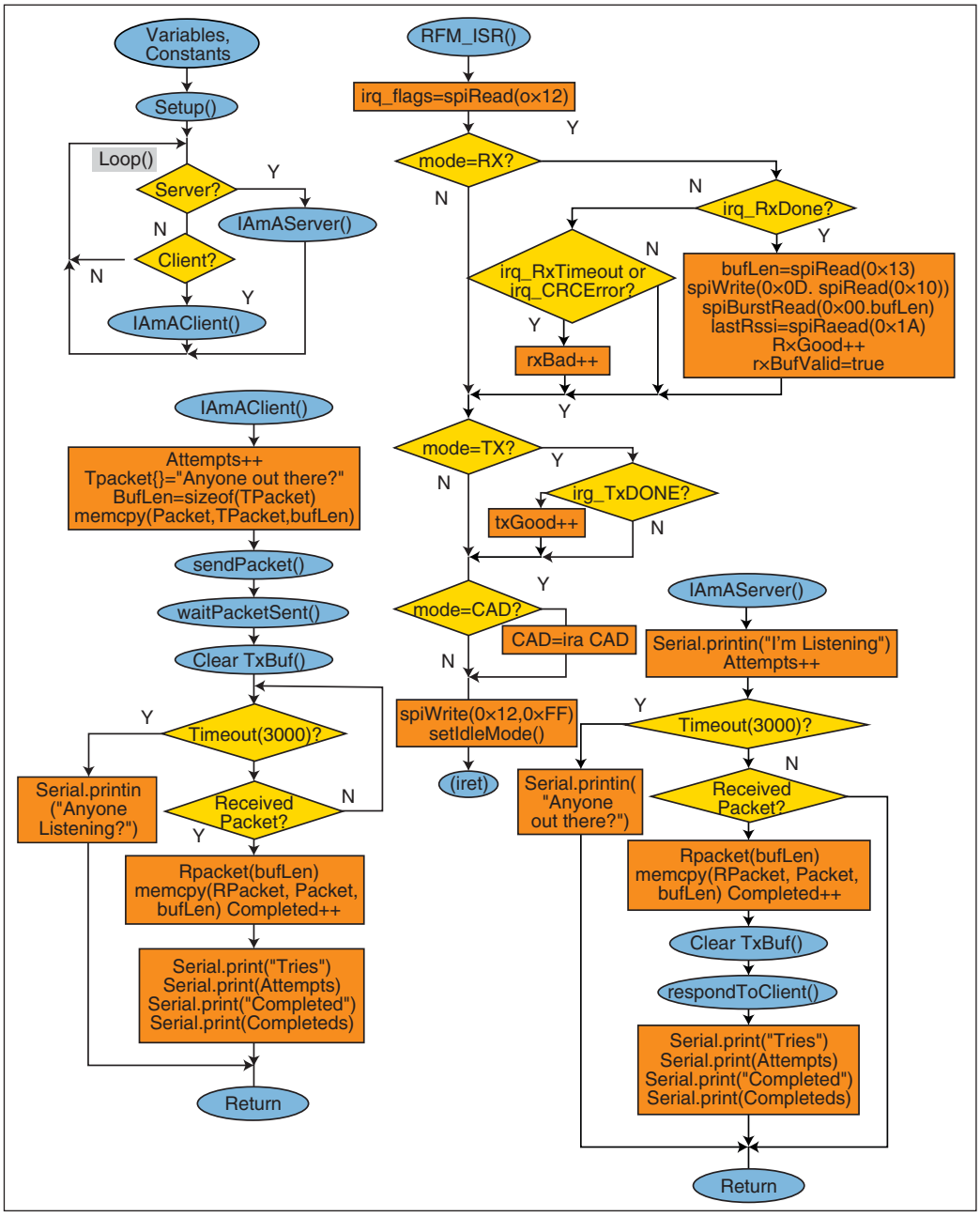

FIGURE 1
This flowchart shows the two basic parts of any Arduino application, setup() and loop(). The preassigned Boolean variable IAmAServer, will enable one of the two functions in loop(), IAmAServer and/or not (IamAClient).

Before leaving the interrupt routine, you should clear the `irq_flags` in register 0x12 and finally change the mode to IDLE. The interrupt routine is enabled in the `setup()` routine, just after all of variables are declared, constants are established, and libraries are included. While you will find RFM95W libraries for use with this module, I am not using any of these so that none of that black magic is hidden from view. While I do use the SPI library, this is not the technology I am trying to present.

The main `loop()` routine determines a function to call dependent on the state of `IamAServer`. This Boolean variable is set to true or false when each Feather 32U4 is programmed to designate the function of the PCB. A server will listen for packets and then reply. A client will send out packets and look for the server's response. If either has been waiting for 3 s, the timeout timer halts the present operation and the program execution proceeds finishing the function and repeating.

Name	Command	Data	Default	Description
Fundamental Command Table				
Set Contrast Control	0x81			
		0x00:FF	0x7F	1 of 256 contrast steps. Contrast increases as the value increases.
Entire Display ON	0xA4:5		0xA4	0 = Ram, 1 = Ignore
Set Normal/Inverse Display	0xA6:7		0xA6	0 = Normal, Inverse
Set Display ON/OFF	0xAE:F		0xAE	0 = Sleep, 1 = Normal
Scrolling Command Table				
Continuous Horizontal Scroll Setup	0x26:7			0 = Right Scroll, 1 = Left Scroll
		0x00		Dummy byte
		0x07:0		Start Page Address
		0x07:0		Scroll Step Time Interval in Frames
		0x07:0		End Page Address
		0x00		Dummy byte
		0xFF		Dummy byte
Continuous Vertical and Horizontal Scroll Setup	0x29:A			0 = Right Scroll, 1 = Left Scroll
		0x00		Dummy byte
		0x07:0		Start Page Address
		0x07:0		Scroll Step Time Interval in Frames
		0x07:0		End Page Address
		0x00		Dummy byte
		0xFF		Dummy byte
Deactivate Scroll	0x2E			Stop scrolling and clear the last scrolling setup command
Activate Scroll	0x2F			Start scrolling the last scrolling setup command
Set Vertical Scroll Area	0xA3			
		0x3F:0	0x00	Number of rows not scrolled
		0x7F:0	0x64	Number of rows scrolled
Addressing Setting Command Table				
Set Lower Column Address (Page Mode)	0x0F:0		0x00	For Page Addressing Mode
Set Higher Column Address (Page Mode)	0x1F:0		0x00	For Page Addressing Mode
Set Memory Address	0x20			Addressing Modes
		0x03:0	0x2=02	0x00 = Horizontal Addressing Mode
				0x01 = Verical Addressing Mode

CLIENT/SERVER

When I go to a restaurant and choose from the menu, I am the restaurant's client requesting food service. The wait staff, or my server, will respond by delivering the entree of choice from the kitchen. Should the staff ask, "How is everything?", they become a client by requesting a response from me. Hopefully, I'll serve a thumbs up to complete the transaction. The client/server role can switch back and forth, depending on who is initiating the transaction. How about that cup of coffee? What if the staff is roaming around with high test in one hand and decaf in the other asking if you want coffee. Is the server the client? Are you the server? It can all get a little confusing.

I like using a browser as an interface. I know when I do I am always the client. When surfing the web, it's clear that, as a client, you request information from Big Brother the server. You may be asked to log in, in which case you are serving up your credentials, but by and large Big Brother has all the info and

				0x02 = Page Addressing Mode
				0x03 = Invalid
Set Column Address	0x21			For Vertical or Horizontal Addressing Mode
		0x3F:0	0x00	Column Start Address
		0x3F:0	0x3F	Column End Address
Set Page Address	0x22			For Page Addressing Mode
		0x07:0	0x00	Page Start Address
		0x07:0	0x07	Page End Address
Set Page Start	0xB7:0			For Page Addressing Mode
Hardware Configuration Command Table				
Set Display Start Line	0x7F:40		0x40	0-63
Set Segment Remap	0xA1:0		0xA0	0 = Column Address 0 to COM 0
				1 = Column Address 0 to COM 127
Set Multiplex Ratio	0xA8			
		0x3F:F		N + 1 (16-64)
Set Output Scan Direction	0xC8:0	0x00		0xC0 = Normal, 0xC8 = reversed
Set Display Offset	0xD3			
		0x3F:0		Verical Shift of COM
Set COM Pins Hardware Configuration	0xDA			
		0x32:02		0x02 = Disable COM remap, Sequential COM
				0x12 = Disable COM remap, Alternative COM
				0x22 = Enable COM remap, Sequential COM
				0x32 = Enable COM remap, Alternative COM
Timing & Driving Scheme Setting Command Table				
Set Display Clock Divide Ratio & Oscillator Frequency	0xD5			
		0xFF:0	0x80	Lower nibble = Divide Ratio
				Upper nibble = Oscillator Frequency
Set Pre-charge Period	0xD9			
		0xFF:0	0x22	Lower nibble = Phase 1 period clocks
				Upper nibble = Phase 2 period clocks
Set VcomH Deselect Level	0xD8			
		0x30:00	0x20	0x00 = 0.65 × VCC
				0x20 = 0.77 × VCC
				0x30 = 0.83 × VCC

TABLE 1

The register set of the SSD1306 OLED driver is similar to bit-mapped LCD drivers. Row and column outputs must be configured to the display it is attached to. This display uses the SSD1306 in its I²C mode. It requires only clock and data lines beside power (5 V/3.3 V) and ground and is addressed at either 0x78 or 0x7A (hardware jumper).

you just want to access it. We are beginning to see a "cloud" of servers sprouting out of necessity. Many offer to hold our precious data, so we can have it available from anywhere anytime, day or night.

When you think Internet of Things (IoT), you may very well put yourself in the role as the client who will access the cloud responsible for serving up a device. Whether this is in the cloud or directly on your own server makes little difference. Those of us who relish direct control will go to extremes to avoid that third-party connection, even if its free (today). Many companies are betting that once you are using their product you'll be willing to pay for its convenience somewhere down the line. You know the line, "The first taste is free."

When all you need is to eliminate a wire, the client server relationship may not be required. Peer-to-peer relationships make sense when the system shares resources. You may remember the pre-2000 MP3 file-sharing service Napster. Napster allowed one to place their music collection on-line to share with likeminded Napster users. This forced copyright infringement talks, which eventually toppled the service as users were personally taken to court. There probably aren't any legal issues sharing the status of the sensors on my weather station. While I might offer these to the public to help the dissemination of local weather info, I don't necessarily want the status (or control of) my porch light open to anyone. Within my home, the sharing of input status and ability to control things is pretty much contained within individual peer-to-peer networks.

In general, there are two trains of thought used in many schemes—the announcement of events and the request of status. An outdoor thermometer might announce, "It's 90°F outside." The living room thermostat might hear this and because it is set to a temperature lower than that, say 80 degrees, it needs to change its potential requests from, "The living room temperature is OK." or "Please warm up the living room.", to "The living room temperature is OK." or "Please cool down the living room.". The Furnace and the Air Conditioning system will hear all this and will determine by the data 'warm' or 'cool', which system is required to service the thermostats request.

The other approach might be to give the furnace and the air conditioning control of its own destiny by request periodic updates from all sensors to determine the correct course of action. You'll note that this second approach requires a lot more communication. For remote battery-powered sensors, this can cause high use of the battery, which leads to premature exchange or recharge. The juggling of too little versus too much communication requires deep thought.

LORA ENDPOINTS

Sensors and actuators can be considered endpoints. They take on a dead-end or an end-of-the-line configuration. While they could become part of a mesh system used to forward information on to others out of range of the first, this project is meant to explore LoRa used in 1:1 communications. This might be peer-to-peer or client/server. I'm going to use the client/server configuration because I'm going to have one endpoint as a base station, which will merely serve back a response whenever it receives a request. The second endpoint is a mobile client that I can move around. This is the where all of the work gets done.

The work here is to make constant requests and keep track of the responses. Since this is a mobile operation, I'll be adding some additional circuitry to the Feather32U4. A couple of push buttons will allow me to raise or lower the transmitter power via a register in the RFM-95W. The button inputs allow me to adjust the TX power if I wanted to trade power for maximum transmission distance. More important for my purposes here, I will add a display to give me visual confirmation of the operational status. I chose to use a tiny 1" OLED bit-mapped display. It uses an I2C interface consisting of just four wires.

This is not as easy to use as a character display, but is worth its weight in gold and therefore deserves some insight as how to use it. Like the RFM-95W, it too has an Arduino library available for it, but I won't be using it here. I'll just write directly to display module so you can get a better feel for how it works. As you can tell from the registers in **Table 1**, it is similar to other displays. The display driver handles a number of

ABOUT THE AUTHOR

Jeff Bachiochi (pronounced BAH-key-AH-key) has been writing for *Circuit Cellar* since 1988. His background includes product design and manufacturing. He can be reached at jeff.bachiochi@imaginethatnow.com or via his website at www.imaginethatnow.com.

interfacing options. While the I2C interface has the fewest number of connections, it is one of the slowest in terms of data transfer.

The display is 128 bits wide by 64 bits high. It is fully bit addressable; however, it is also smartly divided into eight pages. Each page can be thought of as eight rows of 128 columns each. One column within a row (eight pixels) can be written with a single byte. This makes building a text character as easy as transferring 6 bytes—five of character data and one blank column.

Initialization of the display must be done to instruct the driver on how the display is attached. Default values take care of much of this, but there remain a few registers that must be adjusted. You might also note that some registers can be used to rotate the display's orientation, mirror pixel position, and invert all the pixel data. Since pixels are individual LEDs, no backlight is necessary, and pixel brightness is controlled through a register value.

I'm not going to be displaying any pictures on this bit-mapped display, but I am going to be displaying text. In reality, I have to draw pictures of each character as there is no character generator in this display. To help with this, I defined the constant character array, `font[]`, in program memory to preserve RAM space. The array is a list of 5-byte chunks, one byte for each column making up a 5 × 7 character. Since I don't need to display any control characters, the character array begins with character 0x20, the space character (0x00, 0x00, 0x00, 0x00, 0x00). When I want a particular character, say, the lowercase "h," I start with its ASCII value 0x68 and remove the array's lack of the control characters 0x00-0x1F by subtraction 0x20 (0x68-0x20=0x48). I then grab the five data bytes starting from the array's character offset 0x48 × 5 (or 72 × 5 = 240) and send this data to the display.

This is a job I'll be doing often so it gets its own function `selectCharacter()`. Since I'll want to position my text on the display I'll need a function to set `cursorPosition()`. This is handled by writing to the page and column registers. While the page value uses a single register, the column address requires two registers, the column is divided into nibbles with each written to its own address. Once I position the cursor, I pass a message string to the `message()` function which loops through each character of the message and uses `selectCharacter()` to deliver the appropriate data to the display to form the text message one character at a time. A couple of other useful functions were written to help keep the display clean, `clearLine()` and `clearOLED()`.

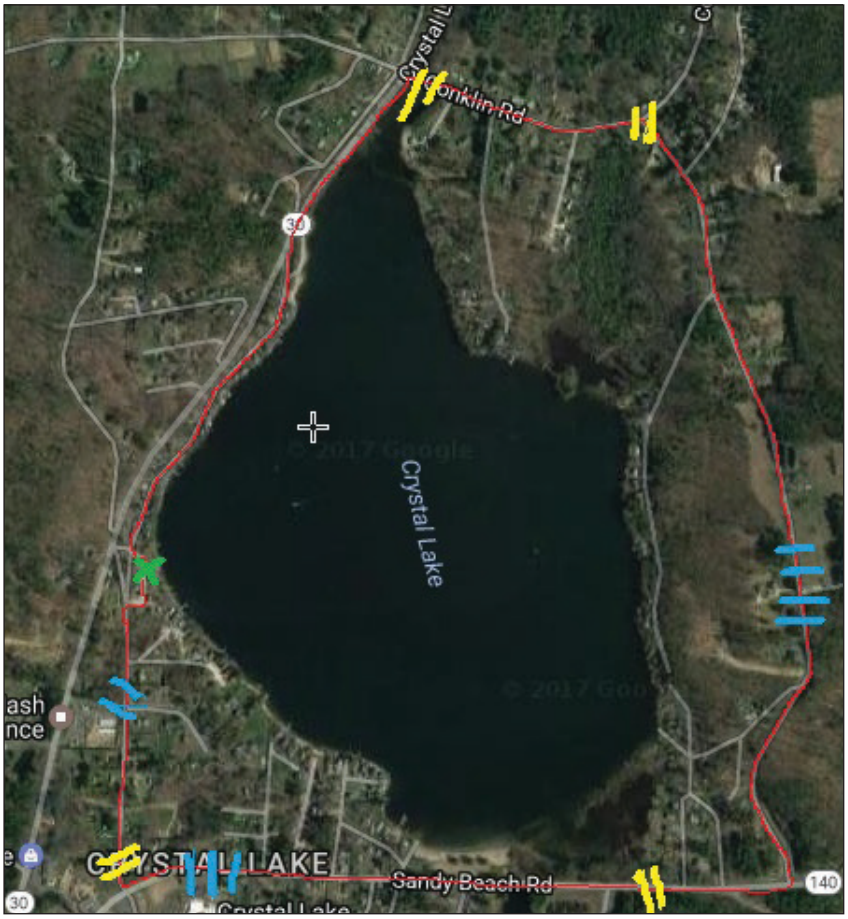

PHOTO 1
The roads surrounding Crystal Lake have widely different terrains giving this experiment a number of different type of interference including, clear line of sight, densely populated housing, forest, hills (above) and dales (below) grade.

FIELD TEST

One Feather32u4 module is connected to my PC as the server. It will remain in my lab listening for a broadcast from another module. For every request it recognizes, it will respond in kind. RealTerm displays any contact, and if none is heard within 3 seconds, it displays a "no contact" message and returns to listen again. Using a timeout lets me know that the program is still executing instead of just a blank screen.

The second Feather32u4 module has the OLED display attached to it and is operated from a 1.8-Ah Li-ion battery (PlayStation remote's battery). I measured about 20 mA as the module's average draw during constant transceiver operations. With the module in hand, I began my walking journey around Crystal Lake. Refer to **Photo 1** for the Google map image of Crystal lake (http://crystal-ct.lakesonline.com/Maps). The journey begins at the green "x" and proceeds along the red line clockwise around the lake area back finally back home. The route is 3.1 miles in length. I've noted low points in yellow and high

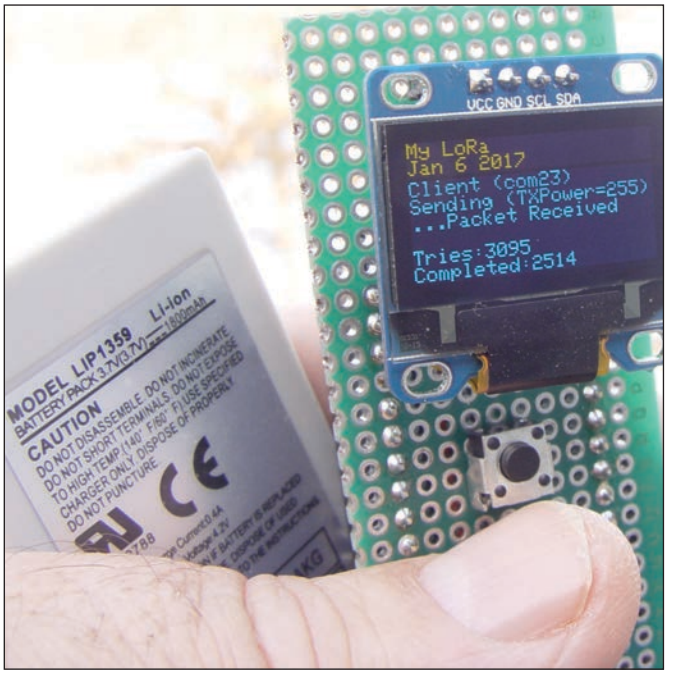

PHOTO 2
The OLED display shows the result of my "hike" around Crystal Lake. The Li-ion battery has enough juice to power this module for approximately four days of constant transmissions. Since the Feather32U4 also has a charger built-in, the USB connection would recharge the battery with little fuss.

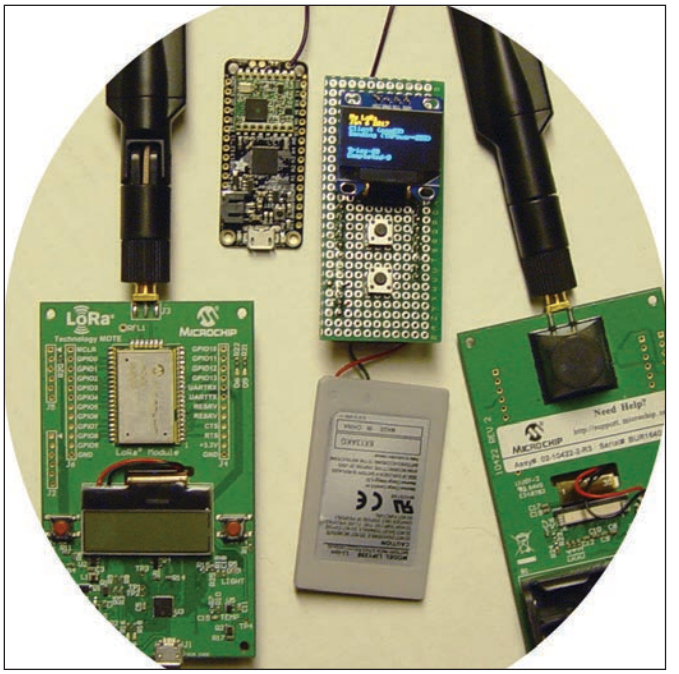

PHOTO 3
On the left and right are the LoRaWAN boards from Microchip. You can compare the LoRa radios modules to the that on the Feather32U4 (top) PCBs. The OLED display (and buttons) are mounted on protoboard attached to the Feather32U4 through their DIP headers.

points in blue to give you a sense of the elevation with yellow around the lake level (636') and blue the highest (825').

From my house down West Shore Road, the houses can be within 50' of each other. I found small areas where I lost momentary contact as there were many buildings between me and home. Once away from houses along Route 30, the signal was continuous. The signal remained strong as I turned onto Conklin Road, where the water flows over the dam at the end of the lake. The clear path across water became quickly hidden by housing and forest and again I began to get drop outs in data. I thought that I might be about at its maximum range as I neared a right on to Minor Road, which climbed to the high point in the area.

On the way to the top of the hill, the signal again returned with gusto. At the peak, housing is sparse as there is mostly forest between me and the lake. At this point, I'm about a mile from my house as the crow flies and the signal is still continuous. As I drop down the back side of Minor Road toward the intersection of Route 140, I pass through an area that is cut through the surrounding hill. Even though I can see layers of rock papering the 100' cliffs exposed by the excavated road, my requests are still being received and answered for the most part. This intersection is the furthest point from my office. Once on Route 140, I begin to see the lake again as I am passing its far shore and the signal remains strong. I get to enjoy the view before reentering another densely populated area. There are areas of missed connections again as I get over the hump and down to my last right back onto West Shore Road. This end of West Shore is a bit hilly, but a straight shot to my house means "good" contact all the way back to the office.

Refer to **Photo 2**. Over the course of the test, the client made 3,095 attempts and a message was received 2,514 times. That's a 81% success rate. I stopped every few feet to check on the "completed" status and found it impossible to find a spot where all the attempts failed.

LoRa technology touts the SX127x family can achieve a range of more than nine miles in a suburban environment and three miles in a dense urban environment. When deployed for metering, security, or industrial automation, range is kept limited to one mile in a suburban environment. I've seen that kind of performance with a simple wire antenna!

circuitcellar.com/ccmaterials

SOURCES

Feather 32u4 Bluefruit LE
Adafruit | www.adafruit.com

RFM95W 868/915Mhz RF Transceiver Module
HOPE Microelectronics | www.hoperf.com

LORAWAN

I've shown that LoRa technology is viable as a communication medium. The LoRa Alliance wants you to buy into this technology as the basis for their grand design. A LoRaWAN network architecture is typically a star topology with a gateway as a transparent bridge relaying messages between end devices and a central network server in the backend. Gateways are connected to the network server via standard IP connections. End devices use single-hop wireless communication to one or many gateways. Communication can be bidirectional and support multicast to reduce the on-air communication time. The LoRaWAN layer targets critical infrastructure and confidential personal data with special emphasis on secure communication through the use of a Unique Network key (EUI64), Unique Application key (EUI64), Device specific key (EUI128), and AES CCM (128-bit) for encryption and authentication.

You'll find many companies offering full tilt solutions that should play well together if they adhere to Alliance standards. I recently purchase a few "Motes," Microchip's DM164139 that feature their RN2903 LoRaWAN modules and a USB connection (see **Photo 3**). These are end devices meant to be used with their LoRaWAN Gateway Radio and Core board to complete the LoRaWAN wireless link with a network server.

While not designed for peer-to-peer (no gateway) LoRaWAN communications, you have access to three command categories via ASCII commands: `sys`, `mac`, and `radio`. As you might guess, the system commands pertain to some basic configuration, the MAC commands with LoRaWAN communications, and radio commands with the LoRa transceiver. With these modules, you can bypass LoRaWAN with a `mac pause` command and then have access the radio directly. Here you may use LoRa technology to send and receive packets using the `radio tx <data>` and `radio rx 0` commands. These modules feature SMA connectors and real antennas and offer one access to both LoRa technology and full LoRaWAN support (with an appropriate gateway).

I invite you to investigate LoRa technology when you find that your requirements exceed the specifications of the wireless technology you are presently using. It may be still too early to tell whether LoRaWAN will be *the* player in our IoT future, but you can be sure the LoRa technology has advantages over some older competitors.

Verilog HDL
With the right tools
designing a microprocessor can be easy.
Okay, maybe not easy, but certainly less complicated. Monte Dalrymple has taken his years of experience designing embedded architecture and microprocessors and compiled his knowledge into one comprehensive guide to processor design in the real world.
Monte demonstrates how Verilog hardware description language (HDL) enables you to depict, simulate, and synthesize an electronic design so you can reduce your workload and increase productivity.
cc-webshop.com
Microprocessor Design Using Verilog HDL
by Monte Dalrymple
elektor